Value-Sensitive Automatic Code Specialization for Embedded Software

Eui-Young Chung, *Student Member, IEEE*, Luca Benini, *Member, IEEE*, Giovanni DeMicheli, *Fellow, IEEE*, Gabriele Luculli, *Member, IEEE*, and Marco Carilli, *Member, IEEE*

***Abstract*—The objective of this work is to create a framework for the optimization of embedded software. We present algorithms and a tool flow to reduce the computational effort of programs, using value profiling and partial evaluation. Such a reduction translates into both energy savings and average-case performance improvement, while preserving a tolerable increase of worst case performance and code size. Our tool reduces the computational effort by specializing frequently executed procedures for the most common values of their parameters. The most effective specializations are automatically searched and identified, and the code is transformed through partial evaluation. Experimental results show that their technique improves both energy consumption and performance of the source code up to more than a factor of two, in average about 35% over the original program. Also, their automatic search engine greatly reduces code optimization time with respect to exhaustive search.**

***Index Terms*—Code size, common value, embedded software, energy consumption, framework, partial evaluation, performance, search space, specialization, value profiling.**

I. Introduction

PROCESSOR-BASED embedded systems are pervasive in many modern application domains such as telecommunications, consumer electronics, and multimedia [5], [6]. The major driving force to move from application-specific to processor-based architectures is programmability, which increases flexibility and reduces the design time. Cost is also reduced, because the design is based on high-volume commodity parts (processor and memory), whereas ASIC solutions require low-volume custom components [2], [3].

The overall performance of processor-based design critically depends on software quality. For this reason, software optimization is one of the most important issues in modern embedded system design [7]–[10]. Embedded software can be optimized more aggressively than applications for general-purpose systems by exploiting detailed knowledge of workloads and hardware platforms. Such optimization is often a critical step for striking design targets under tight cost constraints, which are typical of embedded systems.

A traditional quality metric for embedded software is compactness: the most compact code for a program uses the least instruction memory. Moreover, if such a program represents a pure data flow (i.e., no branching and iteration is involved), it executes in the shortest time and consumes the least energy under the assumption that the cost of each instruction is roughly constant. However, as algorithm complexity grows, the control dependency of a program increases and specific architectural features of a processor may favor some instructions over others in terms of performance and energy consumption. Thus, two additional metrics, namely performance and energy, are considered in embedded software design. It is also very important to distinguish between average and worst case performance because many embedded systems are targeting real-time applications [2].

Average case performance is tightly related to energy efficiency, because short execution time can be directly translated into reduced energy by slowing down the system's clock (or by gating the clock) and/or by lowering the voltage supply [1], [10], [11]. At the same time, however, worst case performance should not be adversely affected when optimizing for average case. In other words, while minimizing the expected value of program execution time, variance should remain under control. In this context, we propose an automatic source code transformation framework aimed at reducing the computational effort (*the average number of executed instructions*) with tightly controlled worst case performance and code-size degradation.

According to Amdahl's law, the most effective way to improve the average case performance is to make the common case fast. Many code transformation techniques exploit execution frequency profiling to identify the most frequently executed code blocks [21], [22] or *computational kernels*. Then, the kernels can be optimized either by eliminating redundant operations or by matching computation and memory transfer to the characteristics of the hardware platform (e.g., parallelizing computation, improving locality of memory transfers) [11], [30], [35].

Procedure inlining is a good example for this concept. This technique first identifies the procedure calls executed frequently then replaces each identified procedure call with a copy of the body of the called procedure, replacing each occurrence of a formal parameter with its corresponding actual parameter [30]. This technique eliminates all the overhead for the procedure invocation, but its primary disadvantage is the code-size increase.

Manuscript received November 16, 2001. This work was supported in part by the National Science Foundation under Grant CCR-9901190, by STMicroelectronics, and by GSRC/MARCO. Recommended by Associate Editor R. Camposano.

E.-Y. Chung and G. DeMicheli are with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA (e-mail: eychung@stanford.edu; nanni@stanford.edu).

L. Benini is with Department of Electrical Engineering and Computer Science, University of Bologna, Bologna 40136, Italy (e-mail: lbenini@deis.unibo.it).

G. Luculli and M. Carilli are with AST, STMicroelectronics, Grenoble 38019, France (e-mail: gabriele.luculli@ast.st.com; marco.carilli@ast.st.com).

Publisher Item Identifier 10.1109/TCAD.2002.801096.

Execution frequencies of program fragments are not the only profiling information that can be used for code optimization. Recently, *value profiling* has been proposed as a technique for identifying a new class of common cases for a given program [23]–[25]. The common cases identified by value profiling are code fragments which frequently execute operations with the same operand values. In this case, the identified code fragments can be specialized for the commonly observed operand values to eliminate redundant computations.

Profiling-driven optimization is often very effective for embedded systems because embedded software can be characterized by a few well-known workloads, unlike software running on a general purpose system. For example, many DSP programs execute filter operations and the filter coefficients are rarely changed. In our framework, procedure calls, which are frequently executed with rarely varying parameter values, are defined as common cases. Such common cases are identified by value profiling and specialized by *partial evaluation.*

Partial evaluation is a transformation technique for specializing a procedure with respect to a subset of its parameters, where these parameters are held constant [4], [18]. Even though partial evaluation is a well-developed field, there are several issues in its application that have not been fully addressed in the past. First, the procedures to be specialized, their parameters, and parameter values for specialization are assumed to be specified by the user. Second, partial evaluation sometimes leads to code-size blowup. If applied in an uncontrolled fashion, it can actually worsen performance and energy consumption. Third, when multiple procedures within a program are specialized, the interplay among various specialized calls is rarely taken into consideration (refer to the example in Fig. 1). Because of these limitations, program specialization based on partial evaluation is not widely applied.

Our source code optimizer automates computational kernel specialization through partial evaluation. The tool integrates execution frequency and value profiling, candidate computational kernel selection, partial evaluation, performance, and energy estimation within a single optimization engine. Its input is a target program (C source code) with typical inputs. The output is optimized source code and estimates of average execution time and energy for the original and optimized version of the target program. The impact of optimization is assessed by instruction-level simulation on the target hardware architecture [12]–[14].

The manuscript is organized as follows. In Section II, we review related work in embedded software optimization with emphasis on value-based specialization techniques. In Section III, we will demonstrate the basic idea and overall flow of the proposed technique for program specialization based on partial evaluation and value profiling. Also, search spaces to be explored are defined. In Section IV, we will present the profiling method and the computational-effort estimation technique. In Section V, common-case selection technique for specialization based on computational-effort estimation will be described. In Section VI, specialization for each common case will be presented and the globally optimal case selection from multiple specialized cases will be discussed in Section VII. Finally, we will report experimental results in Section VIII and conclude our work in Section IX.

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ..............
  result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc, int fk) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
```

(a)

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ..............
  if (cvd_foo(a, k)) result = sp_foo(c);
  else result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc, int fk) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
int sp_foo(int *fc) { return 0; }
int cvd_foo(int a, int k) {
  if (a == 0 && k == 0) return 1;
  return 0;
}
```

(b)

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ..............
  if (cvd_foo(a)) result = sp_foo(c, k);
  else result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
int sp_foo(int *fc, int fk) { return 50*100*fk; }
int cvd_foo(int a) {
  if (a == 0) return 1;
  return 0;
}
```

(c)

Fig. 1. Example of source code transformation using our method: (a) original program; (b) specialized program for the first call of `foo` (`a` = 0 and `k` = 0); and (c) specialized program for the first call of `foo` (`a` = 0).

II. Related Work

The objective of software optimization for general purpose computers is average case performance, while the requirements for embedded software are more articulated [45]. Code-size minimization is often a high-priority objective [28], [29], [50] and energy efficiency is becoming critical as well [53].

Retargetability is a key requirement for embedded software optimization tools, because of the wide variability of target hardware platforms [46], [47], [49], [51]. Also, compiler development for specific application domains such as digital signal processing was researched to exploit the special features of application-specific processor architectures [54], [56]. Most research on optimizing compilers for embedded processors has

focused on fairly low-level optimizations, such as instruction scheduling, register assignment, etc. Embedded software optimization takes advantage of the reduced compilation speed requirement with respect to general-purpose software compilers, therefore embedded software development tools can adopt more complex and aggressive approaches which are not allowed in general purpose software development.

Recently, high-level approaches (based on source-to-source transformations) to improve code quality were proposed. Memory-oriented code transformation techniques were proposed in [15] and [57] and other classical high-level loop transformations for general purpose software were applied to embedded software optimization [12]–[14]. Source-to-source techniques are more aggressive in modifying the target program, and they can be applied together with more traditional optimizing compilers in the back end. One of the major concerns in the adoption of high-level optimizations is that they are hard to control, and they are often meant to be used in a semiautomated flow that requires programmer's guidance.

Value locality is a promising high-level technique for general purpose software optimization, but it has not been studied in depth for embedded software. Value locality is defined as the likelihood of a previously seen value recurring repeatedly within a physical or logical storage location [42]. Value locality enables us to reduce the computational cost of a program by reusing previous computations.

Previous work shows that value locality can be exploited in various ways depending on the target system architecture. In [27], common-case specialization was proposed for hardware synthesis using loop unrolling and algebraic reduction techniques. In [42] and [43], value prediction was proposed to reduce the load/store operations with the modification of general purpose microprocessor. Also, in [34], redundant computation (an operation performs the same computation for the same operand value) was defined and *result cache* was proposed to avoid redundant computations by reusing the result from the *result cache.* Unfortunately, these techniques are not appropriate for our case because they are architecture dependent. For this reason, we will focus on pure software oriented approaches exploiting value locality (i.e., partial evaluation) in this paper.

Depending on the way of using the results of previous computations, partial evaluation can be classified into two categories, i.e., program specialization and data specialization. Program specialization encodes the results of previous computations in a *residual program,* while data specialization encodes these results in the data structures like caches [19].

Program specialization is more aggressive in the sense that it optimizes even the control flow, but it can lead to a code explosion problem due to overspecialization. For example, code explosion can occur when a loop is unrolled and the number of iterations is large. Furthermore, code explosion can degrade the performance of the specialized program due to increased instruction cache misses. On the other hand, data specialization is much less sensitive to code explosion because the previous computation results are stored in a data structure which requires less memory than the textual representation of program specialization. However, this technique should be carefully applied such that the cached previous computations are expensive enough to amortize the cache access overhead. The cache can also be implemented in hardware to amortize the cache access overhead [34].

Our technique is based on program specialization without any hardware assistance for embedded software design. Our approach differs from previous approaches [18] as follows. First, we propose a computational effort estimation technique which combines value profiling with execution frequency profiling. Using the estimation technique, it is possible to identify the common cases (computationally intensive procedure calls with their effective known parameter values for the specialization) in an automated fashion. Second, our approach provides a systematic loop controlling strategy to avoid the code explosion problem (which was manually controlled by the user in previous work). Third, our approach supports the interprocedural effect analysis of the program specialization which was mentioned only in a few papers [17]. This analysis is especially important when multiple procedure calls are specialized.

III. Basic Idea and Overall Flow

A. Basic Idea and Problem Description

The technique described in the following sections aims at reducing the computational effort of a given program by specializing it for situations that are commonly encountered during its execution. The ultimate goal of this technique is to improve energy consumption as well as performance by reducing computational effort. The specialized program requires substantially reduced computational effort in the common case, but it still behaves correctly. The "common situations" that trigger program specialization are detected by tracking the values passed to the parameters of procedures. The example in Fig. 1 illustrates the basic idea.

Consider the first call of procedure `foo` in procedure `main`. Suppose the first parameter `a` is 0 for 90% of its calling frequency. Also, suppose the same condition holds for the last parameter `k`. Using these common values, a partial evaluator can generate the specialized procedure `sp_foo` as shown in Fig. 1(b) which reduces the computational effort drastically.

In reality, the values of parameters `a` and `k` are not always 0. Therefore, the procedure call `foo` cannot be completely substituted by the new procedure `sp_foo`. Instead, we replace it by a conditional statement which selects the appropriate procedure call depending on the result of a *common value detection* (CVD) procedure named `cvd_foo` in Fig. 1(b). We call this transformation step *source code alternation.* Also, the variable whose value is often constant (e.g., `a`) is called *constant-like argument* (CLA).

When the CVD procedure detects a common case, the specialized code corresponding to the detected common case is executed, which yields fewer instruction executions than the original code. On the other hand, the worst case scenario occurs when the CLA does not take any frequently observed values identified by the profiling. In this case, the worst case performance degradation per each call is simply the product of the cost of compare instruction and the number of CLAs tested in the conditional statement, therefore the worst case performance

degradation is marginal if the target procedure is computationally expensive.

In general, different possibilities for code optimization exist. This gives rise to a set of search problems that aim to detect the best set of transformations for the example shown in Fig. 1. If we ignore the common value of k, the original code will be specialized as shown in Fig. 1(c). The `sp_foo` in Fig. 1(c) has one more multiplication than the `sp_foo` in Fig. 1(b), but the situation that $a = 0$ will happen more frequently than the situation that both a and k are 0. For this reason, it is not clear which specialized code is more effective to reduce the overall computational effort. This is the first search problem in our approach.

Next, consider two procedure calls inside the loop of Fig. 1 with the assumption that parameter e (the second parameter of the third procedure call) has single common value, 200. Each of two procedure calls has a CLA as their second argument, respectively. Partial evaluation can be applied for each procedure call to reduce computational effort. However, there is not much to be done by partial evaluator except loop unrolling because all other parameters are not CLAs. The effect of loop unrolling can be either positive or negative depending on the system configuration. For this reason, it is required to find the best combination of loop unrolling for each call. In this example, there are four possible combinations for each call, but the number of combinations is exponential with respect to the number of loops. This is the second search problem of our approach.

After each call is specialized with the best combination of loop unrolling, it is also necessary to check the interplay among the specialized calls, because both specialized calls will increase code size and they may cause cache conflicts due to their alternative calling sequence. Thus, we need a method to analyze the global effect of the specialized calls caused by their interplay, which is the third problem of our approach. This paper addresses these problems and proposes algorithms for the search of the best code specialization.

To summarize, we have three search problems to specialize a program for common cases.

1) **Common case selection** is to find the most effective common case among several common cases for each procedure call.
2) **Common case specialization** is to specialize a procedure call for the given common case by controlling loop unrolling.
3) **Global effective case selection** is to find the most effective combination of specialized calls.

We will use the term "call site" and "procedure call" interchangeably unless there is an explicit explanation. Also, for the sake of simplicity, we will call cycle-accurate instruction-level simulation (simulator) instruction-level simulation (simulator).

B. Framework Configuration and Transformation Flow

The automated framework configuration is shown in Fig. 2, where an instrumentation tool and a profiler provide the basic information necessary to search the solution space. The computational effort estimator solves the common case selection problem and the specialization engine and global effect analyzer solve the common case specialization and the global effective case selection problems, respectively. The entire framework is implemented based on SUIF [36]. CMIX [20] is chosen as a partial evaluator in the specialization engine. The instruction-set level simulator (ISS) in both the specialization engine and global effect analyzer can be selected depending on the target processor to consider the underlying hardware architecture for the specialization. Each tool component in Fig. 2 corresponds to each step of the overall transformation flow shown in Fig. 3. Thus, we will briefly describe each step in this section and the details will be described in the later sections.

- **Instrumentation and profiling:** Two types of profiling are performed, *execution frequency profiling* and *value profiling.* Using the information from *execution frequency profiling,* the computational efforts of procedures and procedure calls are estimated. On the other hand, *value profiling* identifies CLAs and their common values by observing the parameter value changes of procedure calls.
- **Common case selection:** Based on profiling information, all detected common cases are represented as a hierarchical tree (Section V). To reduce the search space, *normalized computational effort* (NCE) is computed for each object in the hierarchical tree. NCE represents the relative importance of each object in terms of computational effort. By defining a user-defined constraint called *computational threshold* (CT), trivial common cases are pruned.
- **Common case specialization:** Each case not pruned in the previous step is specialized. In our framework, specialization is performed by CMIX [20] which is a compile-time (off-line) partial evaluator. In addition to the specialized procedure, the *common value detection* (CVD) procedure is generated. Also, source code alternation is performed so that the original procedure call is replaced by a conditional statement as shown in Fig. 1. For the specialized code of each common case, instruction-level simulation is performed to assess the quality of the specialization and the cases which show improvement by specialization are selected for the next step. The search space of this problem is exponential with respect to the number of loops and the details of heuristic approaches performed by the loop controller for the search space reduction will be described in Section VI.
- **Global effective case selection:** This step analyzes the interplay of the specialized calls chosen at the previous step and decides the specialized calls to be included in the final solution. The search space for this analysis is also exponential with respect to the number of the specialized calls, thus a search space reduction technique based on the branch and bound algorithm is applied to the binary tree built on the specialized calls.

IV. Profiling

A. The Structure of Profiler

Many profiling techniques are based on assembler or binary executables to extract more accurate architecture-dependent information such as memory address tracing and execution time estimation. Since they are designed for specific machine architectures, they have limited flexibility [21].

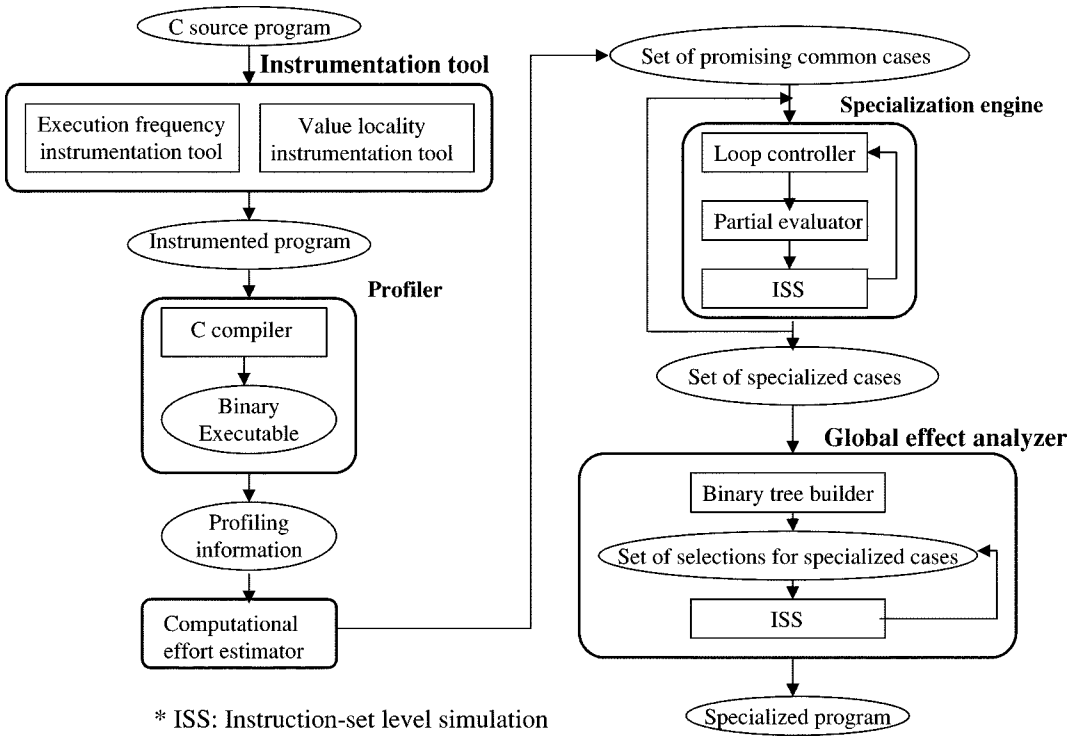


Fig. 2. Configuration of the proposed framework.

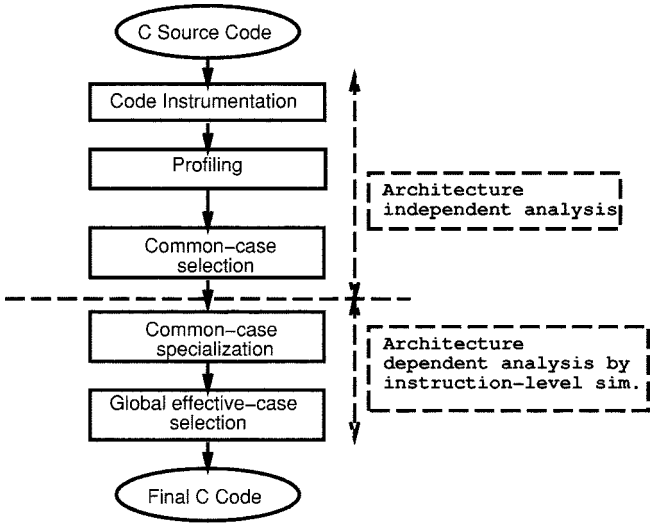


Fig. 3. Overall source code transformation flow.

In our case, it is sufficient to have only relatively accurate information rather than accurate architecture-dependent profiles, while keeping source-level information. In other words, it is more important to identify which piece of code requires the largest computational effort rather than to know the exact amount of computational efforts required for its execution.

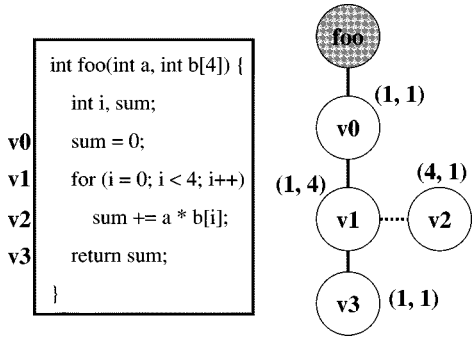
We used the SUIF compiler infrastructure [36] for source code instrumentation. The instrumentation is performed based on the abstract syntax trees (high-SUIF) which will represent the control flow of the given program in high-level abstraction. In detail, a program is represented as a graph $G = \{V, E\}$, where node set V is matched to the high level code constructs such as `for-loop`, `if–then–else`, `do-while` and denoted as $v_i \in V$, $i = \{0, 1, \cdots, N_v - 1\}$, where N_v is the total number of nodes in a program G. Any edge $e_{ij} \in E$ connects two different nodes v_i and v_j and represents their dependency in terms of either their execution order or nested relation. Note that v_i is hierarchical, thus each v_i can have its subtree to represent the nested constructs. For each v_i which is a procedure, we insert as many counters as its descendent nodes to record the visiting frequencies. And for each descendent node, SUIF instructions for incrementing the corresponding counter are inserted for *execution frequency profiling. Value profiling* requires additional manipulations such as type checking between formal parameters and actual parameters of procedure calls, recording the observed values, and so on.

The proposed profiler has the so-called ATOM-like structure [37] in the sense that a user-supplied library is used for instrumentation, namely the source code is instrumented with simple counters and procedure calls. The user-supplied library includes the procedures required for both *execution frequency* and *value profiling.* At the final stage, the instrumented source code and the user supplied library are linked to generate the binary executable for profiling.

B. Computational-Effort Estimation

Computational kernels can be identified by execution frequency profiling and computational-effort estimation. Execution frequency profiling is a widely used technique to obtain the visiting frequency of each node (v_i in G). This information represents how frequently each node is visited, but does not show the importance of each node in terms of computational effort.

For this reason, we used a simple estimation technique of computational efforts for each basic unit using the number of instructions of each basic unit, where the instruction set used is



(a)

Instruction type		o	c
v0	load	1	1
v1	compare	1	1
	increment	1	1
v2	add	1	1
	multiply	1	2
	load	2	1
v3	return	1	2

(b)

Fig. 4. Example of abstract syntax tree and instruction cost table. (a) Example of abstract syntax tree. (b) Corresponding instruction cost table.

the built-in instructions defined in SUIF framework. Due to the lack of specification of a target architecture, it is assumed that all the instructions require same computational effort. But, we provide a way to distinguish the cost of each instruction when the target architecture is determined using an instruction cost table. Each SUIF instruction is defined with its cost in the instruction cost table, thus the execution time of each node v_i of graph G can be calculated as follows:

$$ce_i = f_i * i_i \sum_{j=0}^{N-1} (o_{ij} * c_j) \tag{1}$$

where ce_i is the estimated computational effort of node v_i, f_i is the execution frequency of node v_i from execution frequency profiling, i_i is the average number of iterations for each visit of node v_i, o_{ij} is the number of instructions j observed in node v_i, c_j is the cost of instruction j, and N is the total number of instructions defined in SUIF. Note that the basic unit of our approach includes for–loop and do–while constructs. For this reason, variable i_i is considered in (1). It is also worthwhile to mention that (1) represents the single-level computational-effort estimation. As mentioned in Section IV-A, the node v_i is hierarchical. Thus, the cumulative computational efforts for each node v_i can be estimated by the sum of current level computational effort and the computational effort of its descendent nodes.

An example of abstract syntax tree is shown in Fig. 4(a), where a solid edge represents the dependency of two nodes and a dotted edge represents their nested relation and its corresponding instruction cost table is shown in Fig. 4(b). A pair of numbers assigned to each node is (f_i, i_i) which is obtained from the *execution frequency profiling.*

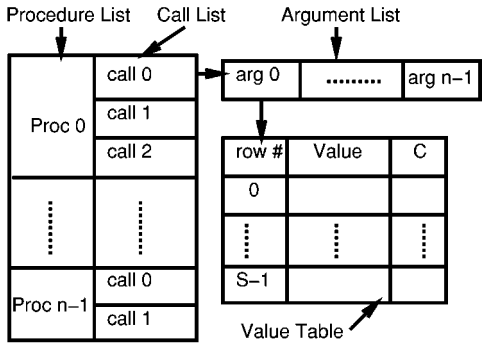


Fig. 5. Internal data structure of value profiling.

Example: Consider node v_2 in Fig. 4(a). f_2 and i_2 are 4 and 1, as shown in the graph. Also, from Fig. 4(b), v_2 has one addition, one multiplication, and two load instructions. Among them, only multiplication has cost twice higher than the other two instruction. By substituting these values into (1), $ce_2 = 4 \times 1 \times (1 \times 1 + 1 \times 2 + 2 \times 1) = 20$. Similarly, $ce_0 = 1$, $ce_1 = 8$, and $ce_4 = 2$. Therefore, the computational effort of procedure `foo` is 31 by summing these values. ■

C. Value Profiling

As mentioned in Section III-A, value profiling is performed at the procedure level. In other words, each procedure call is profiled because any single procedure can be called in many different places with different argument values. We chose value profiling instead of value tracing which records the entire history of value observation because tracing requires huge disk space and accesses.

One of the difficulties in value profiling occurs when the argument size is dynamic. For example, in a program, one-dimensional (1-D) integer arrays with any size can be passed to an integer-type pointer argument whenever the corresponding procedure is called. Another difficulty occurs when the argument has complex data type because complex data type requires hierarchical traversal for value profiling. For this reason, the current value profiling in our work is restricted to elementary-type scalar and array variables. Note that this restriction is not applied to the arguments defined at each procedure but to the variables passed as arguments for each procedure call. When a procedure call has both types of variables as arguments, only the variables which violate this restriction are excluded from profiling. Pointers to procedures are not considered in our approach due to their dynamic nature.

Fig. 5 shows the internal data structure of the value profiling system. As shown in Fig. 5, each procedure has a list of procedure calls which are activated inside the procedure. Each procedure call in the list has a list of arguments and each argument in this list satisfies the type constraint mentioned above and has its own fixed size value table to record the values observed and their frequencies. Each row in the value table consists of three fields: index field, value field, and count (C) field.

The index field represents not only the index of the row, but also the chronological order of the row in terms of the updated time relative to other rows. Thus, the larger the index is, the

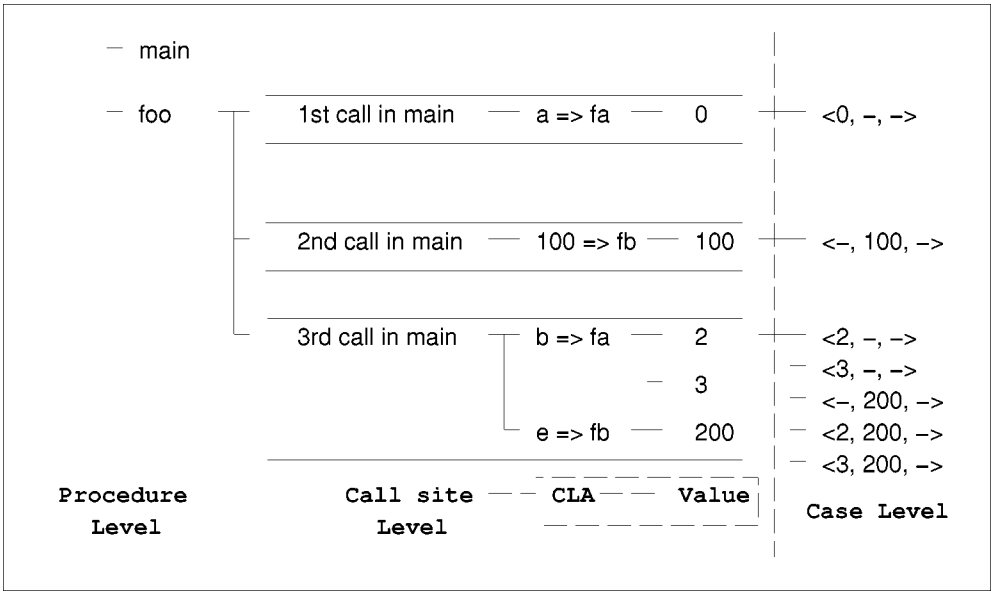


Fig. 6. Hierarchical tree representation of common cases.

more recently the corresponding row is updated. In our representation, each row is denoted as r_i, $i \in \{0, 1, \ldots, S-1\}$, where S denotes the size of value table, i.e., the number of the rows in the table. The value field is used to store the observed value, and the c_i field in r_i counts the number of observations of the corresponding value. The table is continuously updated whenever the corresponding procedure call is executed. At the end of profiling, each argument of the value table is examined to find the values which are frequently observed, and only the argument-value pairs which satisfy user defined constraint called *Observed Threshold* (OT) are reported to the user. For this purpose, *Observed Ratio* (OR_i) is calculated for each r_i in the value table as follows:

$$OR_i = c_i/f \tag{2}$$

where f is the visiting frequency of this call site. The larger OR_i is, the more frequently the value is observed. When OR_i is smaller than OT, the value in r_i is disregarded.

The key feature of value profiling is the value table replacement policy [24]. As mentioned above, the size of each value table is fixed to save memory space and table update time. The variable c_i of each value table is initialized to 0. Thus, if a new value is observed and at least one of c_i is 0, the new value is recorded in r_i which has the smallest index among these rows. On the other hand, when the table is full (there is no c_i which is 0), the following formula is used to select the row which is to be replaced:

$$rf_i = W * \frac{i}{S} + (1 - W) * OR_i \tag{3}$$

where rf_i, $i \in \{0, 1, \ldots, S-1\}$ is replacement factor which is the metric to decide which row is to be replaced. The smaller rf_i is, the more likely r_i will be selected for replacement. The weighting factor W is used to specify the importance of the chronological order relative to observed count c_i. The selected r_i which has the smallest rf_i is deleted from the table and $r_j \rightarrow r_{j-1}, j \in \{i+1, \ldots, S-1\}$ if $j < S-1$. Finally, the new value is stored into a new row r_{S-1}, and thus the table will contain those S entries for which rf_i is largest.

V. Common Case Selection

As shown in the example of Section III-A, any procedure call with CLAs can be specialized. Some procedure calls can be effectively specialized, while others may not show significant improvement. Also, some CLAs are not useful for specialization. Thus, it is necessary to search the procedure calls that can be effectively specialized by using their common values.

Due to the large search space, we represent all possible common cases as a hierarchical tree based on profiling information and prune out the cases which are expected to show only marginal improvement even after specialization.

A. Common Case Representation

Fig. 6 shows the hierarchical tree for the example shown in Fig. 1 based on the profiling information. Let us consider a simple example how a common case is represented in a hierarchical tree. The program has two procedures: `main` and `foo` (*procedure level*), and procedure `foo` is called three times in procedure main (*call-site level*). The first procedure call has single CLA `a` which is passed to the formal parameter `fa` (*CLA level*) and its common value is zero (*value level*). Finally, the common case is represented as $\langle 0, -, - \rangle$ by mapping the common value to the corresponding parameter position (*case level*). For the sake of simplicity, we ignore the parameter `fk` which is the fourth parameter of procedure `foo`. We assume that variable `b` (the first parameter of the third call) has two common values, 2 and 3.

In Fig. 6, the *call-site level* has two-level subhierarchies to represent the CLAs and their common values. *CLA level* represents the mapping relation between CLA parameter and its corresponding formal parameter and *value level* is used for common values of CLAs. In *case level*, common values are related to each formal parameter by positional mapping and

TABLE I
NOTATIONS FOR A HIERARCHICAL TREE

level	set	element
procedure	P	p_i
call site	C_i	c_{ij}
CLA	A_{ij}	a_{ijk}
value	V_{ijk}	v_{ijkl}
case	B_{ij}	$b_{ijm} =< cv_0, \cdots, cv_k, \cdots, cv_{\lvert A_{ij}\rvert-1} >$

"$-$" represents *don't care*—the parameter value in that position is not considered in this case. There are seven possible cases, even though the number of call sites are only three. There is nothing to be examined for procedure `main` because it does not have any CLA.

We introduce some notation for convenience to indicate each level and object in a hierarchical tree as shown in Table I.

As shown in Table I, *procedure level* is denoted as P which is a set of procedures denoted as p_i. Each procedure p_i has a set of procedure calls, $C_i = \{c_{ij}, i = 0, 1, \ldots, |P| - 1, j = 0, 1, \ldots, |C_i| - 1\}$. And the same rule is applied to *CLA level* and *value level*. Each common case of c_{ij} is denoted as b_{ijm} and each dimension of b_{ijm} (cv_k) corresponds to a_{ijk}, $k = \{0, 1, \ldots, |A_{ij}| - 1\}$, where the bound of m ($|B_{ij}|$) will be shown in (4). Also, cv_k of c_{ij} is one of the common values of a_{ijk} or *don't care*, namely $cv_k = \{v_{ijkl}, -\}$, $k \in \{0, 1, \ldots, |A_{ij}|-1\}$, $l \in \{0, 1, \ldots, |V_{ijk}| - 1\}$.

The overall size of the search space to find common cases is the sum of the search space for each call site. At each call site, we need to examine all possible cases with the consideration of the coherence of the common values (the common value of each CLA may occur at the same time or separately). For example as shown in Fig. 1, there are four possible cases: 1) only $a = 0$ (b can have any value); 2) only $b = 0$ (a can have any value); 3) both a and b are zero; 4) neither a nor b is zero (both a and b can have any value). Among these four cases, the last case [case 4)] is ignored due to the lack of useful information for the specialization and total cases to be examined is three. More generally, the search space of each call site, $|B_{ij}|$, is

$$|B_{ij}| = \prod_{k=0}^{|A_{ij}|-1} (|V_{ijk}| + 1) - 1 \tag{4}$$

where $|V_{ijk}| + 1$ represents the number of possible values of each CLA ($+1$ corresponds to any other value except common values) and the last term (-1) represents case 4) (none of the CLAs has a common value).

The overall size of the search space S is

$$S = \sum_{i=0}^{|P|-1} \sum_{j=0}^{|C_i|-1} |B_{ij}|. \tag{5}$$

B. Pruning Trivial Cases

Due to the large size of the common case set, it is necessary to reduce the search space without missing promising candidates. We define *common cases* as those cases to be included in the search space after search space reduction. The search space reduction is performed based on NCE. The computational effort of each procedure is obtained from execution frequency profiling and computational effort estimation technique described in Section IV. Based on this, NCE of each common case can be estimated in a hierarchical order. In other words, NCE of each procedure is estimated first and then NCE of each call site is calculated and so forth.

TABLE II
PROFILING INFORMATION FOR THE HIERARCHICAL TREE SHOWN IN FIG. 6

p_i			c_{ij}				a_{ijk}			
i	proc	NCE	j	site	freq.	NCE	k	var	value	freq.
0	main	5%	0	-	1	5%	-	-	-	-
1	foo	95%	0	1st	100	8%	0	a	0	100
			1	2nd	10000	29%	1	100	100	10000
			2	3rd	10000	54%	0	b	2	1000
									3	8000
							1	e	200	10000

NCE in a hierarchical tree represents the maximum degree of improvement to be obtained by specializing all cases belonging to the given node. For pruning purpose, a user constraint called *computational threshold* (CT) is defined in terms of NCE. We will assume $CT = 0.1$ for all examples illustrated in this section.

Usually, maximizing the usage of common values is considered to be better because more information is provided to the optimizer. But in our case, maximizing the usage of common values is not always advantageous (e.g., the third call in Fig. 1).

Example: Consider two common cases $\langle 2, 200, - \rangle$ and $\langle -, 200, - \rangle$ for the third call of procedure `foo`. The profiling information is shown in Table II which is a sample profiling information used for all examples in this section. From Table II, $b = 2$ with the probability of 0.1 and $e = 200$ with the probability of 1.0. Then, the probability that case $\langle 2, 200, - \rangle$ will happen is 0.1, while that of case $\langle -, 200, - \rangle$ is 1.0. Thus, the specialized code for case $\langle 2, 200, - \rangle$ is useful only when it reduces the computational effort ten times more than the specialized code for case $\langle -, 200, - \rangle$. The cases like case $\langle 2, 200, - \rangle$ are pruned out before progressing to the next step, *i.e., common case specialization*, for the sake of the computation efficiency. ■

Pruning is not limited only to *case level*, but also performed at any other level based on NCE. We will describe NCE computation and pruning at each level in the next subsections.

1) Procedure Level Pruning: NCE of each procedure is obtained by normalizing its computational effort to the total computational effort. Because NCE of procedure `main` is lower than CT, it is eliminated from the hierarchical tree. Also, the procedure which does not have any descendant is eliminated. The pruning at this level has the largest impact on reducing the search space.

2) Call-Site Level Pruning: Call-site level pruning, similar to procedure level pruning, is performed next. The profiler described in Section IV can estimate the computational effort of each procedure as well as each procedure call. Thus, NCE of each procedure call can be computed in the same way as NCE of each procedure is computed. In Table II, the first call of procedure `foo` will be pruned out because its NCE is less than the threshold CT.

We also consider NCE for two subhierarchies in the *call-site level.* NCE of each CLA is calculated by weighting the NCE of the corresponding procedure call (c_{ij}) by its *observed ratio* (OR_i) and can be represented as in (6). Also, NCE of each common value (v_{ijkl}) can be computed similarly

$$NCE(a_{ijk}) = NCE(c_{ij}) * \sum_{k=0}^{|A_{ij}-1|} OR_k. \tag{6}$$

Example: Let us consider the third call of procedure `foo`, where a_{120} is the variable `b` as shown in Table II. a_{120} has two common values, 2 (v_{1200}) and 3 (v_{1201}). Also, from (2), $OR(v_{1200}) = 1000/10\,000 = 0.1$ and $OR(v_{1201}) = 8000/10\,000 = 0.8$. Thus, $NCE(a_{120}) = 0.54 \times (0.1 + 0.8) = 0.486$ which is larger than CT, thus a_{120} is not pruned at *CLA level.* At *value level,* $NCE(v_{1200}) = NCE(a_{120}) \times OR(v_{1200}) = 0.486 \times 0.1 = 0.0486$ which is smaller than CT and v_{1200} is pruned out, whereas v_{1201} is not eliminated because its NCE is larger than CT. ■

3) Case Level Pruning: NCE of each case can be calculated using NCE of common values. But NCE at this level cannot be obtained in the same way used in other levels because each case may depend on multiple common values such as case $\langle 2, 200, - \rangle$. Thus, NCE of each case is obtained by multiplying NCE of common values which are involved in forming the case and represented as follows:

$$NCE(b_{ijm}) = \prod_{k=0}^{|B_{ij}|-1} NCE(cv_k). \tag{7}$$

Remember that cv_k is v_{ijkl}, $l \in \{0, 1, \ldots, |V_{ijk}| - 1\}$ or "—" and NCE$(-)$ is defined as 1. Also, note that NCE is a conservative metric because a case which has a large NCE may not be observed frequently. But this metric is still meaningful in the case level to prune the cases which yield marginal improvements.

Example: Let us consider case $b_{120} = \langle 2, 200, - \rangle$ which is a child of c_{12} (third call in `main`) and c_{12} is also a child of p_1 (procedure `foo`). From the example in Section V-B-2, $NCE(v_{1200}) = 0.0486$. Similarly, $NCE(v_{1210}) = 0.54 \times 10\,000/10\,000 = 0.54$. From (7), $NCE(b_{120}) = 0.0486 \times 0.54 = 0.027$, thus b_{120} is dropped from the search space. But this pruning does not happen in practice because v_{1200} is already pruned out at *value level.* Also, notice that case $\langle -, 200, - \rangle$ which has less information than case $\langle 2, 200, - \rangle$ (from the viewpoint of a specializer in the next step) is still in the tree due to its high NCE (0.54). ■

To reduce the search space further, we define *dominated cases* those that can be eliminated from the search space. We say that b_{ijm} is dominated by b_{ijt} if all common values of b_{ijm} appear in b_{ijt} and $NCE(b_{ijt})$ is greater than or equal to $NCE(b_{ijm})$

$$NCE(b_{ijm}) \leq NCE(b_{ijt}) \qquad \forall\, cv_k \; in \; b_{ijm} \in cv_k \; in \; b_{ijt} \tag{8}$$

where $a \in b$ is defined as *true* when $a = b$ or $a = -$. For example, b_{121} is dominated by b_{124}. A dominated case needs not to be specialized because it has less information and is less important in terms of NCE than dominant case.

To summarize, pruning is performed at each level, but higher level pruning is more effective because its all descendants are removed. Also, notice that pruning sacrifices the amount of the information useful in the specialization step by increasing the possibility that the common situation occurs more frequently (e.g., case $\langle 2, 200, - \rangle$ is pruned, but case $\langle -, 200, - \rangle$ is not). This tradeoff is controlled by the user constraint, CT.

VI. Common Case Specialization

A. Overview

After having pruned out trivial common cases (which show marginal improvement, even when they are specialized), we have only common cases (expected to show nonmarginal improvement by specialization) left in the hierarchical tree. For each remaining case in the hierarchical tree, we perform the specialization using partial evaluation. The common values of each case are used by partial evaluator for: 1) simplifying control flow (precomputing *if* test or unrolling loops); 2) constant folding and propagation; 3) precomputing well-known functions calls such as trigonometric functions, and so on. These optimizations are not performed independently. Indeed, applying one optimization technique can provide a better chance for other techniques to succeed. For example, loop unrolling can provide a better chance to constant propagation/folding by simplifying control dependency and enlarging basic blocks.

Due to such combined effects, it is not easy to estimate the quality of the specialized code analytically. For this reason, this step uses an instruction-set level simulator for the purpose of code quality assessment with the consideration of the underlying hardware architecture. It differs from the common case selection step which performs architecture-independent analysis. Thus, this step takes much longer time than effective case selection step due to specialization and instruction-set level simulation.

Among the techniques mentioned above, loop unrolling should be used most carefully because its side effect (code-size increase) can severely degrade both performance and energy consumption. But in traditional applications of partial evaluation, this fact is not deeply studied, based on the assumption that taking more space will reduce computational effort [18]. This assumption may be true for general systems such as workstations, but may not be true for the resource limited systems such as embedded systems. Therefore, we need to address our second search problem by exploring various loop combinations for unrolling. The size of search space for each case specialization is simply 2^n, where n is the number of loops inside procedure p_i.

In case of an exhaustive search, the specialization of each case is iteratively performed for the overall search space and each iteration requires instruction-set level simulation to assess the specialized code quality. In our framework, loop unrolling can be suppressed by declaring the corresponding loop index variable as a residual variable. It means that the residual variable will not be specialized, henceforth the corresponding loop construct will not be affected by specialization either. Because the search space is exponential with respect to the number of loops, two heuristic approaches are proposed in this section.

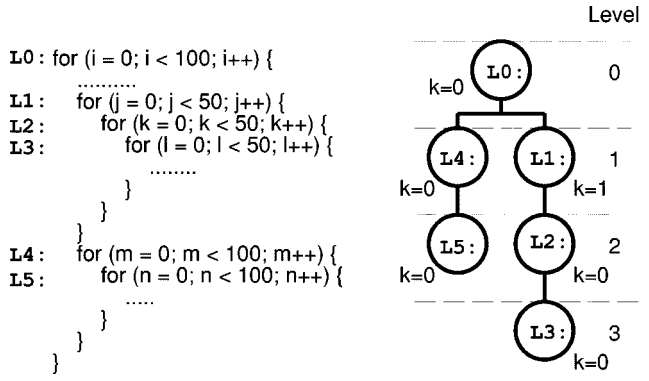


Fig. 7. Example of a loop tree.

The case-level information can be computed from CLA-level information. These two approaches may provide lower quality of specialization over the exhaustive approach, but reduce the search space (both specializations and instruction-set level simulations) drastically.

B. Semi-Exhaustive Approach

The first heuristic search algorithm is called semi-exhaustive search. Unlike pure exhaustive search, semi-exhaustive approach performs a complete search for each loop nest rather than for the entire set of loops. Thus, pure exhaustive search guarantees a globally optimal solution, while the semi-exhaustive approach can provide a suboptimal solution. This is the tradeoff between the searching time and the code quality. The tradeoff will be explained in the experimental part (Section VIII).

For this purpose, we represent the entire loop structure inside a procedure as a loop tree and an example of loop tree is shown in Fig. 7. To construct such a loop tree, we first levelize the loop structure. The outermost loop is assigned to *level 0* and the next outermost loop is assigned to *level 1*, and so on. Next, we represent each loop as a node and place each node to its assigned level. Finally, we represent the nested relation between two nodes as an edge connecting these two nodes. Notice that if a loop has multiple loop nests, the connecting edges are identified as a *branch* and we call each branch path a *subtree.* For example, the edge between $L0$ and $L4$ and the edge between $L0$ and $L1$ form a *branch.* Also, there are two subtrees connected to the branch: a subtree formed by $L4$ and $L5$ and a subtree formed by $L1$, $L2$, and $L3$. Each node is represented as $v_i(k)$, where i is the level to which the node belongs and k is the index of the nodes that have the same parent. Thus, if a node is not connected to a branch, k is always zero.

After constructing a loop tree, the best loop combination for unrolling is searched for each subtree in a bottom-up fashion (i.e., the *branch* in the lower level is visited first). For a given branch, we visit the subtrees in the order of their computational efforts.

While searching the best solution of each subtree, we exclude the loop combinations which are expected to increase the code size drastically, because such loop combinations increase specialization, compilation, and simulation time drastically. Furthermore, such combinations provide a very low quality of specialized code due to the high instruction cache misses. To identify such undesirable cases, we use a code-size constraint and a code-size estimation technique. The code-size constraint is set to the cache size of the target architecture because the code size larger than the cache size will increase the instruction cache miss drastically. Also, the code size is estimated as follows:

$$cs_i(k) = \left(\sum_{j=0}^{|K_{i+1}|-1} cs_{i+1}(j) + NI_i(k) \right) * I_i(k)/U_i(k) \quad (9)$$

where $cs_i(k)$ the cumulated code size of the descendent nodes of node $v_i(k)$ in addition to the code size of $v_i(k)$ itself. Also, $NI_i(k)$ represents the number of instructions of node $v_i(k)$, $I_i(k)$ represents the average number of iterations per each visiting of node $v_i(k)$. Finally, $U_i(k)$ returns 1 when node $v_i(k)$ is unrolled and $I_i(k)$ when $v_i(k)$ is not unrolled. In other words, we estimate the code size to be linearly increased by a factor of $I_i(k)$ when $v_i(k)$ is unrolled. Notice that $I_i(k)$ and $NI_i(k)$ are available from the profiler in Section IV.

Example: Consider the loop tree shown in Fig. 7. Suppose that the subtree on the right *branch* (formed by L1, L2, and L3) has higher computational effort than the subtree on the left *branch* (formed by L4 and L5). In case of a pure exhaustive approach, there are 64 (2^6) combinations of loop unrolling, thus the given case should be specialized and simulated 64 times to find the best combination. In the case of the semi-exhaustive approach, we first visit the right subtree ($L1$) because it has higher computational effort. Because the right subtree is a three-level loop nest (L1, L2, and L3), there are eight combinations of loop unrolling and all combinations are examined to find the best loop combination for the subtree. While examining these eight combinations, the code size of each combination is estimated using (9). If the estimated code size is larger than the code size constraint, the combination is excluded from the specialization. After finding the best combination for the right subtree ($L1$), we visit the left subtree ($L4$) which has four possible loop combinations and find the best solution in the same way. After loop unrolling for both subtrees is decided, we move to the top node ($L0$). There are only two combinations for this node because loop unrolling for all its descendent nodes is already decided. Thus, we need to examine the total 14 loop combinations using the semi-exhaustive approach. ■

C. One-Shot Approach

The second heuristic approach to solve the common case specialization problem is called *one-shot approach.* It is close to the *semi-exhaustive approach,* but differs because the choice of the best combination for each subtree depends on just code-size estimation instead of an exhaustive search within the subtree. The code-size estimation is performed in *depth-first search* fashion for each subtree. We will illustrate this approach using the following example.

Example: Let us consider the loop tree shown in Fig. 7. The subtree ($L1$) is visited first due to the same reason as in the semi-exhaustive approach (higher computational effort). Initially, all nodes are assumed not to be unrolled. However, at this time, all eight possible combinations are not examined. Instead, unrolled code size is estimated in *depth-first order* [from the lowest level

```
main() {
  int *a, b, *c;
  ........
  for (i = 0; i < 100; i++) {
    ........
    if (a[i] > 0)
      foo(a, b);
    ........
    bar(a, c);
  }
}
bar(int *fa, int *fc) {
  bar2(fa, fa[0]);
  ........
  bar2(fc, fc[0]);
}
```

Fig. 8. A more complex example for global effective case selection.

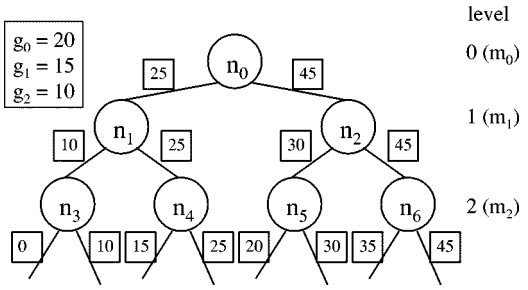


Fig. 9. Example of binary tree for M.

($L3$) to the highest level ($L1$)]. First, $L3$ is visited and the unrolled code size is estimated. If the unrolled code size is larger than the code-size constraint, the code estimation procedure is terminated and the node is decided not to be unrolled. Also, all nodes in the higher level of this subtree are determined not to be unrolled. Otherwise (estimated code size is smaller than code-size constraint), we decide to unroll this node and move up to node $L2$. The same procedure is repeated until it reaches to the top of the subtree. After all nodes in the right graph are traversed, we move to the left graph and the same decision procedure is applied. Finally, we move up to the top node and the same procedure is repeated. ■

To summarize, this approach requires only single specialization and simulation, but it is more limited in improving the quality of partial evaluation.

VII. Global Effective Case Selection

The last search problem is to analyze the interplay among the specialized calls to maximize the specialization effect in a global perspective. We already described this problem in Section III using a simple example in Fig. 1. We consider now a more complex example.

Example: Consider the situation in Fig. 8. Suppose that the call of procedure `foo` and both calls of procedure `bar2` inside procedure `bar` are computationally expensive and have common cases. Then, all three call sites are specialized independently in the common case specialization step. If we analyze their interplay in a local scope (intraprocedural analysis), two calls inside procedure `bar` will interfere with each other marginally. Furthermore, the interplay between procedure call `bar2` and procedure `foo` is not detected because their interplay occurs in interprocedure level, even though they may affect to each other severely. Thus, the interplay among the specialized calls should be analyzed in a global scope (interprocedural analysis). ■

The interprocedural analysis may reveal that the combination of multiple specialized calls may yield a gain inferior to the sum of the gains of the individual specialized calls, because of mutual interference such as the I-cache conflict. Also, it is not obvious to estimate their interference analytically. For this reason, each combination should be assessed by instruction-set level simulation and the best combination is chosen for the final solution.

We represent each specialized call as $m_k \in M$, $k = \{0, 1, \ldots, |M| - 1\}$. Each m_k has an attribute called gain g_k which is the amount of improvement in terms of the given cost metric (either energy consumption or performance) and obtained when each call is specialized at the common case specialization step. We always sort m_ks in descending order for g_k, i.e., $g_k \geq g_{k+1}$. We denote a combination of the specialized calls as $c_i \in C$, $i = \{0, 1, \cdots, |C| - 1\}$ and $|C| = 2^{|M|}$, thus the search space is exponentially large. Each c_i is a binary vector to represent which specialized calls are included in this combination. For example, $c_0 = \langle 1, 1, 1\rangle$ means m_0, m_1, and m_2 are included in the combination c_0. Also, $c_1 = \langle 1, 1, 0\rangle$ means only m_0 and m_1 are included in the combination c_1. Each c_i has two gain attributes $\text{ideal\_}g_i$ and $\text{actual\_}g_i$ which are *ideal gain* and *actual gain*, respectively.

- **Ideal gain ($\text{ideal\_}g_i$)** is the sum of gains of the individual specialized calls in each combination by assuming that there is no interference with each other. Thus, this is the maximum gain that can be achieved for the given combination.
- **Actual gain ($\text{actual\_}g_i$)** is the sum of gains of specialized calls in each combination with the consideration of their interference. Thus, it is always less than or equal to (when there is no interference) the ideal gain and can be obtained by instruction-set level simulation.

We represent each combination c_i as a path in a binary tree as shown in Fig. 9. The rightmost path represents $c_0 = \langle 1, 1, 1\rangle$ and the second rightmost path represents $c_1 = \langle 1, 1, 0\rangle$, and so on. Each level of the tree corresponds to each element of the vector c_i and the right edge and the left edge correspond to "1" and "0," respectively. Thus, the number of levels in the binary tree is always $|M|$. Each edge $e_i(l)$, $i = \{0, 1, \ldots, 2^{(l+1)} - 1\}$ and $l = \{0, 1, \ldots, |M| - 1\}$ also has a gain attribute $g_i(l)$. Where l is the level to which the edge belongs and i is the index of an edge in level l (from left to right).

Initially, $g_i(|M| - 1)$ (the gain of each edge connected to the leaf nodes) is set to $\text{ideal\_}g_i$. At the same time, $g_i(l)$ is set to $\max\{g_{2i}(l + 1), g_{2i+1}(l + 1)\}$, namely the edges above the leaf-level inherit the maximum gain of their children. After the gain initialization as shown in Fig. 9, we perform the search procedure based on the *branch and bound* algorithm in Fig. 10. We will illustrate the how the procedure works using the following example.

Example: The gain for the right edge of n_6 called $g_7(2)$, is initially set to 45 ($\text{ideal\_}g_0$) because this path corresponds to $c_0 = \langle 1, 1, 1\rangle$ which means m_0, m_1, and m_2 are included in the combination c_0. Similarly, $g_6(2)$ (the gain for the left edge of n_6) is set to 35 (corresponds to $c_1 = \langle 1, 1, 0\rangle$). Also, $g_3(1) =$

```
search_solution (binary_tree) {
  initialze_gain;
  solution = traverse_tree(root_node_of_binary_tree);
  return solution;
}
traverse_tree(binary_tree_node) {
  if (binary_tree_node == NULL) {
    simulate the selected case;
    return sim.  result;
  }
  if (right_node_visited == FALSE) {
    select right_edge;
    gain_right_edge = traverse_tree(right_node);
  }
  if (gain_right_edge >= gain_left_edge) {
    delete left descendent; /* pruning */
    if (solution == NULL)
      save current case to solution;
    return gain_right_edge;
  }
  if(left_node_visited)
    return gain_left_edge;
  else {
    select left_edge;
    gain_left_edge = traverse_tree(left_node);
    if (gain_right_edge >= gain_left_edge) {
      select right_edge;
      save this case to solution
      delete left descendent;
      return gain_right_edge;
    }
    else {
      delete right descendent;
      save current case to solution;
      return gain_left_edge;
    }
  }
}
```

Fig. 10. Search procedure for the given binary tree.

$\max\{g_6(2),\ g_7(2)\} = 45$ and the gains of other edges are also decided in the same way. Next, we apply the procedure in Fig. 10. First, we visit the rightmost path (c_0). For c_0, we perform instruction-set level simulation to get $\mathrm{actual\_}g_0$, and $g_7(2)$ is updated to $\mathrm{actual\_}g_0$. We compare $g_7(2)$ to $g_6(2)$ which is the maximum gain that can be achieved by combination c_1. If $g_7(2) \geq g_6(2)$, it is obvious that c_0 is better than c_1, thus we eliminate the left edge of n_6 (identical to eliminate c_1). On the other hand, if $g_7(2) < g_6(2)$, c_1 can be better than c_0. Thus, we perform instruction-set level simulation for c_1 and update $g_6(2)$ with $\mathrm{actual\_}g1$. Then, we can decide which combination is better and prune out the worse combination. Next, we move to node n_2 in the next level by updating $g_3(1)$ to $\max\{g_6(2),\ g_7(2)\}$ without simulation because we already selected either c_0 or c_1 in level 2. If $g_3(1) \geq g_2(1)$, we can prune out the left descendent of n_2 (c_2 and c_3) due to the same reason. But, if $g_3(1) < g_2(1)$, we visit node n_5 to choose the better combination from c_2 and c_3 by performing the same procedure as we did for c_0 and c_1. After choosing either c_2 or c_3, we compare two edges of node n_2 and select better one. We repeat the same procedure until there remains only one path in the binary tree. ■

To summarize, the algorithm first builds a binary tree to enumerate all possible selections of specialized calls. Second, the expected gain of each path is computed as a cost function for the pruning purpose by ignoring the interplay effect. Third, the actual cost of each path is defined as an actual gain considering the interplay effect; this is available from instruction-set level simulation. The purpose of this search problem is to find the path which shows the maximum actual gain among all paths. The pruning occurs when the expected gain of the current path is less than the maximum actual gain obtained up to this point which is the bounding function of this search problem. As a final remark, this step can be extended to consider the code-size increase constraint by the use of the code-size increase estimation mentioned in Section VI.

VIII. Experimental Results

A. Experimental Setting

Even though source code transformations are applicable to a wide set of architectures, we consider now two specific hardware platforms to be able to quantify the results. The SmartBadge, an ARM processor-based portable device [26] and ST200 processor developed by STMicroelectronics and Hewlett-Packard [38], [39] were selected as the target architectures. For these target architectures, we applied the proposed technique to seven DSP application C programs—`Compress`, `Expand`, `Edetect`, and `Convolve` from [32], `g721 encode` from [31], and `FFT` from [33], `FIR` [39], `turbo code` [41], and `SW radio`.

`Compress` compresses a pixel image by a factor of 4:1 while preserving its information content using DCT and `Expand` performs the reverse process using IDCT. `Edetect` detects the edges in a 256 gray-level pixel image using Sobel operators and `Convolve` convolves an image relying on 2-D-convolution routine. `g721 encoder` is a CCITT ADPCM encoder. `FFT` performs FFT using Duhamel–Hollman method for floating-point type complex numbers (16 point). `turbo code` is an iterative (de)coding algorithm of 2-D systematic convolutional codes using log-likelihood algebra. Finally, `SW radio` performs a series of operations (CIC lowpass filter, FM demodulation, IIR/FIR deemphasis) for the input in ADC format.

The experiment was conducted for two aspects, search space reduction and quality of the transformed code. The quality of transformed code was analyzed in terms of energy saving, performance improvement, and code-size increase. Each application program was profiled to collect computational effort and CLAs with their common values. There exist two important parameters in value profiling as described in Section IV-C. First, *observed ratio* (OR) is the ratio of the observation frequency of a specific parameter value over the total call site visiting frequency. Second, *observed threshold* (OT) is a threshold value to select common values among observed parameter values—only the observed parameter values which show OR higher than OT are selected as common values. In this experiment, OT was set to 0.5, thus only observed parameter values which have OR higher than 0.5 were selected as common values.

B. Search Space Reduction

We first analyzed the effectiveness of the proposed search space reduction techniques. Fig. 11 shows the pruning ratio achieved by each step with computation threshold $CT = 0.1$. Notice that this step is architecture-independent as shown in Fig. 3, thus Fig. 11 is common to both SmartBadge and ST200 processor.

The procedure pruning step always plays an important role to reduce the search space, but the call-site pruning step shows large variation depending on the property of the application programs. This is because the computational kernels of some programs such as `compress` and `FFT` were called only once, while the kernel of `g721 encode` was called several times in

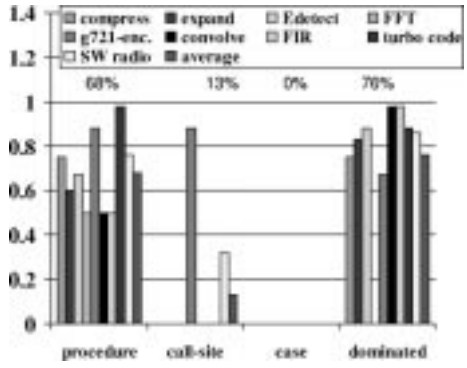


Fig. 11. Search space reduction using common case selection.

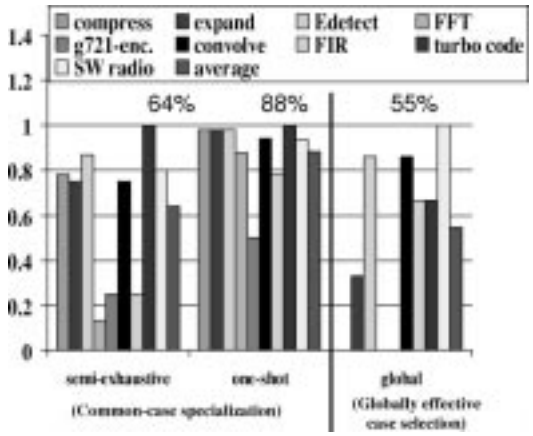


Fig. 12. Search space reduction ratio in common case and global effective case selection step.

different sites with different calling frequencies. Thus, this step is useful for the kernels called frequently in different sites with different frequencies.

The ineffectiveness of the case pruning step was due to high OT which was set to 0.5 for value profiling. Under this OT, the OR of each common value was usually large enough to yield NCE larger than CT used in this experiment (0.1). Dominated case pruning was effective for most of application programs because many of common values were constant ($OR = 1.0$).

Next, the pruning methods used in common case specialization and global effective case selection were evaluated. Fig. 12 shows the pruning ratios of these two steps for SmartBadge environment. Our technique in ST200 processor environment also showed similar results. As shown in Fig. 12, both semi-exhaustive and one-shot approaches drastically reduced the search space by 64% and 88%, respectively. Also, the pruning technique in the global effective case selection step showed 55% of search space reduction and large variation of pruning ratio depending on the property of application programs. There was nothing to be pruned for `Compress`, `FFT`, and `g721 encode` programs because only one case was passed from common case specialization step.

C. Code Quality Improvement

Both the one-shot and semi-exhaustive approaches were compared to the exhaustive approach in terms of code quality and specialization time. The common case selection step was commonly used for each approach to avoid large search space. Also, the global effective case selection step was used in all three specializations because it is an exact solution. As expected, the one-shot approach showed the smallest running time and semi-exhaustive approach was ranked at second. In average, both the one-shot and semi-exhaustive approaches are about 8.3 (8.0) times and 2.7 (2.5) times faster than exhaustive approach in SmartBadge (ST200 processor) environment, respectively.

Notice that Fig. 12 only shows the reduction ratio of the search space, which is different from the specialization time in the sense that search space reduction ratio only implies the reduction ratio of the number of specializations, while the specialization time includes partial evaluation, compilation, and instruction-set level simulation.

In SmartBadge environment, our tool was executed on a SUN UltraSPARC running at 200 MHz with 512-MB memory. The overall procedure of our tool takes less than 20 min with the one-shot approach, while it takes from 10 min (FIR) to 7 h (turbo code) depending on the complexity of the loop structure in addition to the overall program complexity and program input data size.

In ST200 environment, our tool was executed on a Sony VAIO R538DS equipped with a Pentium III running at 500 MHz with 128-MB memory. The difference of execution time between the one-shot and semi-exhaustive approaches is still large, but the semi-exhaustive approach benefits by the faster machine (also faster simulator) because it requires more iterations including simulation, compilation, and specialization. `turbo code` with the semi-exhaustive approach still showed the longest execution time (2 h and 20 min).

It is interesting that the exhaustive approach often generated a huge size of code which is one of the main problems in partial evaluation. For the code, compilation or simulation was not terminated within a few hours, which is a bottleneck for automation. For this reason, we adopted time-out approach especially for the exhaustive approach by assuming that the code requiring long simulation time would be very huge and require large energy consumption.

Table III shows the quality of transformed code in terms of energy, performance, and code size for the three approaches. Notice that the energy consumption was measured with the consideration of shutdown technique. As shown in Table III, the semi-exhaustive approach is comparable to the exhaustive approach in terms of transformed code quality with much less computation time (63% for SmartBadge and 60% for ST200 processor). The one-shot solution is also useful by trading off its code quality and computation time. (About 8.0 times faster and consumes 2% more energy compared to exhaustive approach). We could not perform the exhaustive approach for `turbo code` because its computational kernel had too many loops (18) which yielded a huge number of loop combinations ($2^{18} = 261\,844$). It is also worthwhile to mention that the deviation of improvement is largely depending on the nature of the programs. For the best case, the improvement is more than twice (`Edetect`), but for the worst case, about 10% (0%) is improved (`Compress`) in SmartBadge (ST200 processor) environment.

It is interesting that our tool specialized `Compress` and `Expand` in different ways depending on the target architecture.

TABLE III
QUALITY OF THE CODE TRANSFORMED WITH DIFFERENT APPROACHES (NORMALIZED TO ORIGINAL CODE)

(a) Specialized code quality in SmartBadge environment

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	E	P	S	E	P	S	E	P	S
Compress	0.91	0.91	1.01	0.91	0.91	1.01	0.93	0.93	1.15
Expand	0.84	0.83	1.15	0.84	0.83	1.15	0.90	0.90	1.12
Edetect	0.44	0.37	1.20	0.44	0.37	1.20	0.44	0.37	1.20
FFT	0.86	0.86	1.16	0.86	0.86	1.16	0.86	0.86	1.16
g721 encode	0.88	0.88	1.04	0.88	0.88	1.04	0.88	0.88	1.04
Convolve	0.54	0.48	1.18	0.54	0.48	1.18	0.54	0.48	1.18
FIR	0.53	0.53	1.12	0.53	0.53	1.12	0.53	0.53	1.12
turbo code	-	-	-	0.89	0.90	1.22	0.89	0.90	1.22
SW radio	0.67	0.65	1.09	0.67	0.65	1.09	0.67	0.65	1.09
Average	0.71	0.69	1.12	0.73	0.71	1.13	0.74	0.72	1.14

(b) Specialized code quality in ST200 processor environment

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	E	P	S	E	P	S	E	P	S
Compress	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Expand	0.94	0.95	1.08	0.94	0.95	1.08	1.00	1.00	1.00
Edetect	0.27	0.26	1.04	0.27	0.26	1.04	0.27	0.26	1.04
FFT	0.18	0.19	1.14	0.18	0.19	1.14	0.18	0.19	1.14
g721 encode	0.91	0.95	1.01	0.91	0.95	1.01	0.91	0.95	1.01
Convolve	0.65	0.68	1.04	0.65	0.68	1.04	0.65	0.68	1.04
FIR	0.38	0.35	1.06	0.38	0.35	1.06	0.38	0.35	1.06
turbo code	-	-	-	0.82	0.82	1.23	0.82	0.82	1.23
SW radio	0.81	0.79	1.10	0.81	0.79	1.10	0.81	0.79	1.10
Average	0.64	0.64	1.06	0.67	0.67	1.09	0.68	0.67	1.08

* E: energy, P: performance, S: code size

TABLE IV
IMPROVEMENT RATIO OF FLOATING POINT AND FIXED-POINT VERSIONS (SEMI-EXHAUSTIVE)

programs	SmartBadge		ST200 processor	
	E	P	E	P
Compress_float	0.91	0.91	1.0	1.0
Compress_fixed	0.80	0.79	0.91	0.91
Expand_float	0.84	0.83	0.94	0.95
Expand_fixed	0.55	0.53	0.73	0.76

* E: energy, P: performance, S: code size

`Compress` and `Expand` show nonmarginal improvement in SmartBadge environment, whereas their improvement ratio in the ST200 processor is marginal. Also, the improvement ratio of `FFT` is much larger in ST200 processor environment than in SmartBadge environment, even though the specialized programs for both architectures are identical. The common feature of these programs is that the computational kernels of all three programs have floating-point operations which are not directly supported by the hardware in both architectures, but they are handled by floating point emulation. From the careful analysis of these programs, we found two reasons for this fact. First, the computation cost of floating-point emulation in a ST200 processor is much more expensive than in a SmartBadge environment (relative to their integer operations). Notice that the floating point emulation is performed by the built-in library functions which is out of the scope in our technique. Second, the loop overhead in SmartBadge is larger than in ST200 processor.

The results in Table IV support this claim. `Compress_float` and `Expand_float` are the floating point versions used in Table III and `Compress_fixed` and `Expand_fixed` are their fixed-point versions, respectively. Notice that the improvement by the specialization is mainly due to loop unrolling for both versions of two programs.

As shown in Table IV, the improvement ratio using our technique is about 2.5 times larger for the fixed-point version compared to the floating point version in SmartBadge environment. On the other hand, it is about five times larger in ST200 processor environment. It means that the relative cost of floating point emulation in ST200 processor environment is twice larger than that in SmartBadge environment. But, the improvement ratio using our technique in SmartBadge environment is still larger than in ST200 processor environment. It implies that the loop overhead elimination by our technique is more effective (about twice) in SmartBadge environment rather than in ST200 processor environment.

In the case of `FFT`, the specialization step eliminates trigonometric functions such as `cos`. The computation cost of a `cos` function is four times more expensive in ST200 processor environment than in SmartBadge environment in terms of number of clock cycles (measured by simulators). Thus, the elimination of such functions is more advantageous in ST200 processor than in SmartBadge environment.

In summary, our technique is more effective in fixed-point arithmetic programs, therefore it is desirable to apply our technique after transforming the floating point arithmetic programs into the fixed-point arithmetic programs as proposed in [40]. Also, the computation cost of the built-in functions such as trigonometric functions is architecture dependent, thus the impact of the specialization varies largely depending on the underlying hardware architecture.

As a final remark, the run time of the optimization flow depends on two user-defined constraints CT and OT that drive the pruning. Also, program size and loop depth are critical factors in specialization step, because our approach uses instruction set-level simulation. Nevertheless, it is important to remember that low energy and fast execution of the target code is the overall objective, which can be achieved at the expense of longer optimization time for large programs.

D. Input Data Sensitivity Analysis

The variation of improvement (whatever the metric is) is a common problem of profiling-based techniques because profiling information can be largely varied depending on the trained input data set. Our technique is also affected by input data set, and the improvement ratio shown in Section VIII-C may be largely varied if common values used for transformation heavily depend on input data set.

We analyzed the common values identified by our framework and they can be classified into two categories. The common values in the first category are sensitive to input data set, while those in the second category are independent to input data set, i.e., they are statically declared (or computed) values somewhere in the program. We call the common values in the first category *dynamic common values* and those in the second category *static common values.* Notice that *static common values* are rarely (or never) changed input data identified by a programmer, but this information is not used for optimization due to the complexity and/or future modification.

A program transformed using *static common values* shows a constant improvement ratio because the transformation is independent to input data set. In our experiment, many of benchmark programs were transformed using *static common values.* `Compress` and `Expand` programs initially compute `cos` table with a fixed number of sampling points and these results are identified as common values. In `g721 encode` program, the *static common values* are a quantization table and its size defined in a program. It is interesting that two quantization tables with different sizes are defined in this program, but only one quantization table was consistently used in each call site. Thus, even though many *static common values* were observed in procedure point of view, each call site was related to a single *static common value.* Programs `Edetect`, `Convolve`, and `FIR` identified filter coefficient tables as *static common values* (with their size) and these coefficient values and sizes were efficiently used for the transformation. In program `turbo code`, the number of delay elements was identified as a *static common value.*

In case of `FFT`, the number of sampling points was identified as a `dynamic common value`, thus the improvement ratio was largely varied. In the worst case, the transformed program does not show any improvement if the identified common value is not observed during the program execution. However, the variation of *dynamic common value* was limited to several numbers such as 4, 8, 16, and so on. Such a limited divergence can be handled in our framework because our framework can manipulate multiple common cases for single call sites using a multiway branch statement with multiple *common value detection* procedures at the expense of code-size increase.

To summarize, our technique shows a constant improvement ratio when it transforms programs with *static common values*, but transformation with *dynamic common values* can largely change the improvement ratio depending on the input data like other profiling-based techniques. Also, restricted variation of *dynamic common value* can be treated by our framework at the expense of code-size increase.

IX. Conclusion

We presented algorithms and a tool flow to reduce the computational effort of software programs, by using value profiling and partial evaluation. We showed that the average energy and run time of the optimized programs is drastically reduced. The main contribution of this work is the automation of an optimization flow for software programs. Such a flow operates at the source level and is compatible with other software optimization techniques, e.g., loop optimizations and procedure in-lining.

Within our approach, a first tool performs program instrumentation and profiling to collect useful information for transformations, such as execution frequency and commonly observed values at each call site. Using the profiling information, another tool selects common cases based on the estimated computational effort. Each selected case is specialized independently using a partial evaluator. In the selection step, code explosion due to loop unrolling—which may hamper partial evaluation—is avoided by code-size estimation technique and pruning. Finally, the interplay among the multiple specialized cases is analyzed based on instruction-set level simulation. The transformed code shows an average 35% (26%) energy savings and 38% (31%) in average performance improvement with 7% (13%) code-size increase in ST200 processor (SmartBadge) environment.

Currently, our approach is limited to the common cases at the procedure-call level, but we believe that our technique can be extended to the lower level common cases (e.g., loop level) which may provide a better quality of code specialization. Also, the specialization technique will be extended to consider more architecture-dependent characteristics.

References

[1] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. New York: Kluwer, 1997.
[2] Y.-T. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*. New York: Kluwer, 1999.
[3] W. Wolf, *Computers as Components—Principles of Embedded Computing System Design*. Boston, MA: Morgan Kaufmann, 2001.
[4] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
[5] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. Paulin, "Embedded software in real-time signal processing systems: Design technologies," *Proc. IEEE*, vol. 85, pp. 436–454, Mar. 1997.
[6] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded software in real-time signal processing systems: Application and architecture trends," *Proc. IEEE*, vol. 85, pp. 419–435, Mar. 1997.
[7] J. R. Lorch and A. J. Smith, "Software strategies for portable computer energy management," *IEEE Personal Commun.*, vol. 5, pp. 60–73, June 1998.
[8] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower," in *Proc. ISCA—Int. Symp. Computer Architecture*, 2000, pp. 95–106.
[9] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. IEEE Symp. Low Power Electronics*, 1994, pp. 38–39.

[10] J. M. Rabaey and M. Pedram, Eds., *Low-Power Design Methodologies*. New York: Kluwer, 1996.
[11] L. Benini and G. De Micheli, "System-level power optimization techniques and tools," *Proc. ACM TODAES—Trans. Design Automation Electronic Systems*, vol. 5, no. 2, pp. 115–192, 2000.
[12] H. Mehta, R. Owens, M. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proc. ISLPED—Int. Symp. Low Power Electronics and Design*, 1997, pp. 72–75.
[13] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Memory system energy: Influence of hardware-software optimizations," in *Proc. ISLPED—Int. Symp. Low Power Electronics and Design*, 2000, pp. 244–246.
[14] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. DAC—Design Automation Conf.*, 1997, pp. 188–193.
[15] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. New York: Kluwer, 1998.
[16] F. Catthoor, S. Wuytack, E. De Greef, L. Nachtergaele, and H. De Man, "System-level transformation for low power data transfer and storage," in *Low-Power CMOS Design*, A. Chandrakasan and R. Brodersen, Eds. New York: IEEE Press, 1998.
[17] K. Cooper, M. Hall, and K. Kennedy, "A methodology for procedure cloning," *Computer Languages*, vol. 19, no. 2, pp. 105–117, 1993.
[18] C. Consel and O. Denvy, "Tutorial notes on partial evaluation," in *Proc. ACM Symp. Principles of Programming Languages*, 1993, pp. 493–501.
[19] S. Chirokoff and C. Consel, "Combining program and data specialization," in *Proc. ACM SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, 1999, pp. 45–59.
[20] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, Univ. Copenhagen, May 1994.
[21] J. Pierce, M. Smith, and T. Mudge, "Instrumentation tools," in *Fast Simulation of Computer Architectures*, T. Conte and C. Gimarc, Eds. New York: Kluwer, 1995, pp. 47–86.
[22] T. Ball and J. Larus, "Optimally profiling and tracing programs," in *Proc. ACM Symp. Principles Programming Languages*, 1992, pp. 59–70.
[23] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 259–269.
[24] ——, "Value profiling and optimization," *J. Instruction-Level Parallelism*, vol. 1, 1999.
[25] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 270–280.
[26] T. Simunic, L. Benini, and G. De Micheli, "Cycle accurate simulation of energy consumption in embedded systems," in *Proc. DAC—Design Automation Conf.*, 1999, pp. 867–872.
[27] G. Lakshminarayana, A. Raghunathan, K. Khouri, K. Jha, and S. Dey, "Common-case computation: A high-level technique for power and performance optimization," in *Proc. DAC—Design Automation Conf.*, 1999, pp. 56–61.
[28] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, "Instruction selection using binate covering for code size optimization," in *Proc. ICCAD—Int. Conf. Computer-Aided Design*, 1995, pp. 393–399.
[29] K. D. Cooper and P. Schileke, "Non-local instruction scheduling with limited code growth," in *Proc. ACM SIGPLAN Workshop Languages, Compilation, and Tools Embedded Systems*, 1998, pp. 193–207.
[30] D. Bacon, S. Graham, and O. Sharp, "Compiler transformation for high-performance computing," *ACM Computing Surv.*, vol. 26, no. 4, pp. 345–420, 1994.
[31] C.-H. Lee. [Online]. Available: http://www.cs.ucla.edu/~leec/mediabench.
[32] M. Stoodley. [Online]. Available: http://www.eecg.toronto.edu/~stoodla/benchmarks.
[33] P. Duhamel and H. Hollman, "Split-radix FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, 1984.
[34] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Sun Microsystems Lab., 1992.
[35] M. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA: Addison-Wesley, 1996.
[36] Stanford Compiler Group, "The SUIF Library: A set of core routines for manipulating SUIF data structures," Stanford Univ., 1994.
[37] A. Srivastava and A. Eustace, "ATOM: A system for building customized programming analysis tools," *Proc. PLDI—Programming Language Design and Implementation*, pp. 196–205, 1994.
[38] . [Online]. Available: http://www.st.com
[39] P. Faraboschi and F. Homewood, "ST200: A VLIW architecture for media-oriented applications," in *Microprocessor Forum*, 2000.
[40] T. Simunic, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proc. ISSS—Int. Symp. System Synthesis*, 2000, pp. 193–198.
[41] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, Apr. 1996.
[42] M. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction," *Proc. ASPLOS—Architectural Support Programming Languages and Operating Systems*, pp. 138–147, 1996.
[43] K. Lepak and M. Lipasti, "On the value locality of store instructions," in *Proc. ISCA—Int. Symp. Computer Architecture*, 2000, pp. 182–191.
[44] V. Nirkhe and W. Pugh, "Partial evaluation of high-level imperative program languages with applications in hard real-time systems," in *Proc. ACM Symp. Principles Programming Languages*, 1992, pp. 269–280.
[45] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*. New York: Kluwer, 1995.
[46] A. Sudarsanam, "Code optimization libraries for retargetable compilation for embedded digital signal processors," Ph.D. disseration, Princeton Univ. Dept. EE, 1998.
[47] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "CodeSyn: A retargetable code synthesis system," in *Proc. Int. Symp. High-level Synthesis*, 1994.
[48] P. Chou and G. Borriello, "Software scheduling in the co-synthesis of reactive real-time systems," in *Proc. DAC—Design Automation Conf.*, 1994, pp. 1–4.
[49] R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems Tools and Applications*. New York: Kluwer, 2001.
[50] R. Leupers, *Code Optimization Techniques for Embedded Processors Methods, Algorithms, and Tools*. New York: Kluwer, 2000.
[51] C. Liem, *Retargetable Compilers for Embedded Core Processors Methods and Experiences in Industrial Applications*. New York: Kluwer, 1997.
[52] F. Thoen, M. Cornero, G. Goossens, and H. De Man, "Software synthesis for real-time information processing systems," in *Proc. Workshop Languages, Compilers, and Tools for Real-Time Systems*, 1995, pp. 60–69.
[53] V. Tiwari, S. Malik, and A. Wolfe, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing Syst.*, vol. 13, no. 1–2, pp. 223–233, 1996.
[54] S. P. Rajan, A. Sudarsanam, and S. Malik, "Development of an optimizing compiler for fujitsu fixed-point digital signal processor," in *Proc. CODES—Int. Workshop Hardware/Software Codesign*, 1999, pp. 2–6.
[55] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the Aviv retargetable code generator," in *Proc. DAC—Design Automation Conf.*, 1998, pp. 510–515.
[56] G. Araujo and S. Malik, "Code generation for fixed-point DSPs," *Proc. ACM TODAES—Trans. Design Automation Electronic Systems*, vol. 3, no. 2, pp. 136–161, 1998.
[57] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *Proc. ACM TODAES—Trans. Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.
[58] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low-power embedded system design," in *Proc. DAC—Design Automation Conf.*, 2000, pp. 294–299.

Eui-Young Chung (S'99) received the B.S. and M.S. degrees in electronic engineering from Korea University, Seoul, Korea, in 1988 and 1990, respectively. He is currently pursuing the Ph.D. degree in electrical engineering, Stanford University.

From 1990 to 1997, he was a Research Engineer in the CAD group, Samsung Electronics, Seoul, Korea. His research interests include CAD of VLSI circuits and system-level low-power design methodology including software optimization.

Luca Benini (S'94–M'97) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1997.

He is an Associate Professor in the Department of Electrical Engineering and Computer Science (DEIS), University of Bologna. He also holds visiting researcher positions at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, CA. His research interests include all aspects of CAD of digital circuits, with special emphasis on low-power applications, and in the design of portable systems. On these topics he has published more than 140 papers in international journals and conferences.

Dr. Benini is a member of the organizing committee of the International Symposium on Low Power Design. He is a member of the technical program committee of several technical conferences, including the Design Automation Conference, International Symposium on Low Power Design, and the Symposium on Hardware–Software Codesign.

Giovanni DeMicheli (S'80–M'82–SM'83–F'94) is Professor of Electrical engineering, and by courtesy, of computer science at Stanford University, Stanford, CA. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware–software codesign, and low-power design. He is author of *Synthesis and Optimization of Digital Circuits,* (New York: McGraw-Hill, 1994) and co-author and/or co-editor of five other books and of over 250 technical articles. He is a member of the technical advisory board of several EDA companies, including Magma Design Automation, Coware, and Aplus Design Technologies. He was member of the technical advisory board of Ambit Design Systems. He is a founding member of the ALaRI institute at Universita' della Svizzera Italiana (USI), in Lugano, Switzerland, where he is currently scientific counselor.

Dr. De Micheli is a Fellow of ACM. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He is President-Elect of the IEEE CAS Society for 2002 and he was its Vice President (for publications) in 1999 through 2000. He was Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS from 1987 to 2001. He was the Program Chair and General Chair of the Design Automation Conference (DAC) from 1996 to 1997 and 2000, respectively. He was the Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively.

Gabriele Luculli (M'01) received the laurea degree and Ph.D. degree in electronic engineering from Pisa University, in 1996, and Scuola Superiore S.Anna of Pisa, in 2000, respectively.

He is a R&D Manager in the Advanced System Technology group of STMicroelectronics, where he contributes to the development of design methodologies for SOC design. From 1995 to 2000, he had several work experiences in different research centers, among them IEI Institute of CNR, NATO Research Center, and PARADES Center in Rome. His research interests include several aspects of design technologies for system-on-chip design, with particular emphasis on system-level design, software performance estimation, and embedded OS synthesis.

Marco Carilli (M'00) graduated in physics at University La Sapienza of Rome, Italy.

He is Director of Design Systems in the Advanced System Technology group of STmicroelectronics. He joined ST in 1987 and he has been working primarily on design automation R&D projects ever since, contributing to the birth of corporate CAD in the company. He has taken various technical and management responsibilities in Central R&D first, then in the former Programmable Products Group (now CMG), where he was responsible for the System-On-Chip R&D group and for setting up and developing the entire corporate design reuse standardization program. Before ST, Marco worked for SOGEI (Finsiel Group).